An Introduction to OCaml

Ronghui Gu

Spring 2019

Columbia University

* Course website: https://www.cs.columbia.edu/ rgu/courses/4115/spring2019

** These slides are borrowed from Prof. Edwards.

An Endorsement?

A PLT student accurately summed up using OCaml:

> *Never have I spent*
> *so much time*
> *writing so little*
> *that does so much.*

I think he was complaining, but I'm not sure.

Other students have said things like

> *It's hard to get it to compile, but once it compiles, it works.*

Why OCaml?

- It's Great for Compilers
  I've written compilers in C++, Python, Java, and OCaml, and it's much easier in OCaml.
- It's Succinct
  Would you prefer to write 10 000 lines of code or 5 000?
- Its Type System Catches Many Bugs
  It catches missing cases, data structure misuse, certain off-by-one errors, etc. Automatic garbage collection and lack of null pointers makes it safer than Java.
- Lots of Libraries and Support
- Lots of Support

OCaml in One Slide

Apply a function to each list element; save results in a list

```
# let rec map f = function
    [] -> []
  | head :: tail ->
      let r = f head in
      r :: map f tail;;

val map : ('a -> 'b) -> 'a list -> 'b list

# map (function x -> x + 3) [1;5;9];;

- : int list = [4; 8; 12]
```

"Is recursive"

Passing a function

Case splitting

Pattern Matching

Polymorphic

Local name declaration

Types inferred

List support

Recursion

Anonymous functions

The Basics

Hello World in OCaml: Interpret or Compile

Create a “hello.ml” file:

```
print_endline "Hello World!"
```

Run it with the interpreter:

```
$ ocaml hello.ml
Hello World!
```

Run it with the bytecode interpreter:

```
$ ocamlc -o hello hello.ml
$ ocamlrun hello
Hello World!
```

On most systems, the bytecode can be run directly:

```
$ ocamlc -o hello hello.ml
$ ./hello
Hello World!
```

Hello World in OCaml: Interpret or Compile

Compile a native executable and run:

```
$ ocamlopt -o hello hello.ml
$ ./hello
Hello World!
```

Use ocamlbuild: built-in compilation rules for OCaml projects handle all the nasty cases; automatic inference of dependencies, parallel compilation, etc.

```
$ ocamlbuild hello.native
$ ./hello.native
Hello World!
```

Hello World in OCaml: REPL

The interactive Read-Eval-Print Loop

```
$ ocaml
        OCaml version 4.02.3

# print_endline "Hello World!";;
Hello World!
- : unit = ()
# #use "hello.ml";;
Hello World!
- : unit = ()
# #quit;;
$
```

Double semicolons ;; mean "I'm done with this expression"

#quit terminates the REPL

Other directives enable tracing, modify printing, and display types and values. Use ledit ocaml or utop instead for better line editing (history, etc.)

Comments

OCaml

```ocaml
(* This is a multiline
   comment in OCaml *)

(* Comments
    (* like these *)
   do nest
*)

(* OCaml has no *)
(* single-line comments *)
```

C/C++/Java

```c
/* This is a multiline
   comment in C */

/* C comments
      /* do not
   nest
 */

// C++/Java also has
// single-line comments
```

Basic Types and Expressions

```
# 42 + 17;;
- : int = 59

# 42.0 +. 18.3;;
- : float = 60.3

# 42 + 60.3;;
Error: This expression has type
float but an expression was
expected of type int

# 42 + int_of_float 60.3;;
- : int = 102

# true || (3 > 4) && not false;;
- : bool = true

# "Hello " ^ "World!";;
- : string = "Hello World!"

# String.contains "Hello" 'o';;
- : bool = true

# ();;
- : unit = ()

# print_endline "Hello World!";;
Hello World!
- : unit = ()
```

Integers (31-bit on 32-bit processors)

Floating-point numbers

Floating-point operators must be explicit (e.g., +.)

Only explicit conversions, promotions (e.g., int_of_float)

Booleans

Strings

The unit type is like "void" in C and Java

Standard Operators and Functions

+ - * / mod	Integer arithmetic
+. -. *. /. **	Floating-point arithmetic
ceil floor sqrt exp log log10 cos sin tan acos asin atan	Floating-point functions
not && ||	Boolean operators
= <> == !=	Structual comparison (polymorphic) Physical comparison (polymorphic)
< > <= >=	Comparisons (polymorphic)

Structural vs. Physical Equality

==, != Physical equality compares pointers

```
# 1 == 3;;
- : bool = false

# 1 == 1;;
- : bool = true

# 1.5 == 1.5;;
- : bool = false   (* Huh? *)

# let f = 1.5 in f == f;;
- : bool = true

# 'a' == 'a';;
- : bool = true

# "a" == "a";;
- : bool = false   (* Huh? *)

# let a = "hello" in a == a;;
- : bool = true
```

=, <> Structural equality compares values

```
# 1 = 3;;
- : bool = false

# 1 = 1;;
- : bool = true

# 1.5 = 1.5;;
- : bool = true

# let f = 1.5 in f = f;;
- : bool = true

# 'a' = 'a';;
- : bool = true

# "a" = "a";;
- : bool = true
```

Use structural equality to avoid headaches

If-then-else

$$\text{if } \mathit{expr}_1 \text{ then } \mathit{expr}_2 \text{ else } \mathit{expr}_3$$

If-then-else in OCaml is an expression. The *else* part is compulsory, expr_1 must be Boolean, and the types of expr_2 and expr_3 must match.

```
# if 3 = 4 then 42 else 17;;
- : int = 17
# if "a" = "a" then 42 else 17;;
- : int = 42
# if true then 42 else "17";;
This expression has type string but is here used with type int
```

Naming Expressions with *let*

let *name* = $expr_1$ in $expr_2$	Bind *name* to $expr_1$ in $expr_2$ only
let *name* = *expr*	Bind *name* to *expr* forever after

```
# let x = 38 in x + 4;;
 - : int = 42

# let x = (let y = 2 in y + y) * 10 in x;;
- : int = 40

# x + 4;;
Unbound value x

# let x = 38;;
val x : int = 38

# x + 4;;
- : int = 42

# let x = (let y = 2) * 10 in x;;
Error: Syntax error: operator expected.

# let x = 10 in let y = x;;
Error: Syntax error
```

Let is Not Assignment

Let can be used to bind a succession of values to a name. This is not assignment: the value disappears in the end.

```
# let a = 4 in
  let a = a + 2 in
  let a = a * 2 in
  a;;
- : int = 12

# a;;
Unbound value a
```

This looks like sequencing, but it is really data dependence.

Let is Really Not Assignment

OCaml picks up the values in effect where the function is defined. Global declarations are not like C's global variables.

```
# let a = 5;;
val a : int = 5
# let adda x = x + a;;
val adda : int -> int = <fun>

# let a = 10;;
val a : int = 10
# adda 0;;
- : int = 5          (* adda sees a = 5 *)

# let adda x = x + a;;
val adda : int -> int = <fun>
# adda 0;;
- : int = 10         (* adda sees a = 10 *)
```

Functions

Calling Functions

C/C++/Java

```
// This is C/C++/Java code
average (3, 4);
```

OCaml

```
(* This is OCaml code*)
average 3.0 4.0
```

no brackets and no comma between the arguments

the syntax average (3.0, 4.0) is meaningful: call the function with ONE argument has the type pair

Defining Functions

C/C++/Java

```
double average (double a, double b)
{
  return (a + b) / 2;
}
```

OCaml

```
let average a b =
  (a +. b) /. 2.0
```

type inference

no implicit casting

no return keyword, the last expression becomes the result

Functions

A function is just another type whose value is an expression.

```
# fun x -> x * x;;
- : int -> int = <fun>
# (fun x -> x * x) 5;; (* function application *)
- : int = 25
# fun x -> (fun y -> x + y);;
- : int -> int -> int = <fun>
# fun x y -> x + y;;   (* shorthand *)
- : int -> int -> int = <fun>
# let plus = fun x y -> x + y;;
val plus : int -> int -> int = <fun>
# plus 2;;
- : int -> int = <fun>
# plus 2 3;;
- : int = 5
# let plus x y = x + y;; (* shorthand *)
val plus : int -> int -> int = <fun>
```

Let is Like Function Application

$$\text{let } \textit{name} = \textit{expr}_1 \text{ in } \textit{expr}_2$$

$$(\text{fun } \textit{name} \text{ -> } \textit{expr}_2)\ \textit{expr}_1$$

Both mean "$\textit{expr}_2$, with *name* replaced by $\textit{expr}_1$"

```
# let a = 3 in a + 2;;
- : int = 5
# (fun a -> a + 2) 3;;
- : int = 5
```

Semantically equivalent; let is easier to read

Recursive Functions

OCaml

```
let rec gcd a b =
  if a = b then
    a
  else if a > b then
    gcd (a - b) b
  else
    gcd a (b - a)
```

C/C++/Java

```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b)
      a -= b;
    else
      b -= a;
  }
  return a;
}
```

let rec allows for recursion

Use recursion instead of loops

Tail recursion runs efficiently in OCaml

Recursive Functions

By default, a name is not visible in its defining expression.

```
# let fac n = if n < 2 then 1 else n * fac (n-1);;
Unbound value fac
```

The *rec* keyword makes the name visible.

```
# let rec fac n = if n < 2 then 1 else n * fac (n-1);;
val fac : int -> int = <fun>
# fac 5;;
- : int = 120
```

The *and* keyword allows for mutual recursion.

```
# let rec fac n = if n < 2 then 1 else n * fac1 n
  and fac1 n = fac (n - 1);;
val fac : int -> int = <fun>
val fac1 : int -> int = <fun>
# fac 5;;
- : int = 120
```

First-Class and Higher Order Functions

First-class functions are treated as values: name them, pass them as arguments, return them

```
# let plus5 x = x + 5;;
val plus5 : int -> int = <fun>
# let appadd f n= (f 42) + n;;
val appadd : (int -> int) -> int -> int = <fun>
# appadd plus5;;
- : int -> int = <fun>
# let appadd5 = appadd plus5;;
val appadd5 : int -> int = <fun>
# appadd5 17;;
- : int = 64
```

Higher-order functions: functions that work on other functions

Tuples, Lists, and Pattern Matching

Tuples

Pairs or tuples of different types separated by commas.

Very useful lightweight data type, e.g., for function arguments.

```
# (18, "Adam");;
- : int * string = (18, "Adam")
# (18, "Adam", "CS");;
- : int * string * string = (18, "Adam", "CS")
# let p = (18, "Adam");;
val p : int * string = (18, "Adam")
# fst p;;
- : int = 18
# snd p;;
- : string = "Adam"
# let trip = (18, "Adam", "CS");;
val trip : int * string * string = (18, "Adam", "CS")
# let (age, _, dept) = trip in (age, dept);;
- : int * string = (18, "CS")
```

Records

OCaml supports records much like C's *structs*.

```
# type stu = {age : int; name : string; dept : string };;
type stu = { age : int; name : string; dept : string; }
# let b0 = {age = 18; name = "Adam"; dept = "CS" };;
val b0 : stu = {age = 18; name = "Adam"; dept = "CS"}
# b0.name;;
- : string = "Adam"
# let b1 = { b0 with name = "Bob" };;
val b1 : stu = {age = 18; name = "Bob"; dept = "CS"}
# let b2 = { b1 with age = 19; name = "Alice" };;
val b2 : stu = {age = 19; name = "Alice"; dept = "CS"}
```

Lists

```
(* Literals *)
[];;                    (* The empty list *)
[1];;                   (* A singleton list *)
[42; 16];;              (* A list of two integers *)
(* cons: Put something at the beginning *)
7 :: [5; 3];;           (* Gives [7; 5; 3] *)
[1; 2] :: [3; 4];;      (* BAD: type error *)
(* concat: Append a list to the end of another *)
[1; 2] @ [3; 4];;       (* Gives [1; 2; 3; 4] *)
(* Extract first entry and remainder of a list *)
List.hd [42; 17; 28];;  (* = 42 *)
List.tl [42; 17; 28];;  (* = [17; 28] *)
```

The elements of a list must all be the same type.

:: is very fast; @ is slower—$O(n)$

Pattern: create a list with cons, then use *List.rev*.

Some Useful List Functions

Three great replacements for loops:

List.map f [a1; ... ;an] = [f a1; ... ;f an]

Apply a function to each element of a list to produce another list.

```
# List.map (fun a -> a + 10) [42; 17; 128];;
- : int list = [52; 27; 138]
# List.map string_of_int [42; 17; 128];;
- : string list = ["42"; "17"; "128"]
```

List.fold_left f a [b1; ...;bn] = f (...(f (f a b1) b2)...) bn

Apply a function to a partial result and an element of the list to produce the next partial result.

```
# List.fold_left (fun sum e -> sum + e) 0 [42; 17; 128];;
- : int = 187
```

Some Useful List Functions

List.iter f [a1; ...;an] = begin f a1; ... ; f an; () end

Apply a function to each element; produce a unit result.

```
# List.iter print_int [42; 17; 128];;
4217128- : unit = ()
# List.iter (fun n -> print_int n; print_newline ())
   [42; 17; 128];;
42
17
128
- : unit = ()
# List.iter print_endline (List.map string_of_int [42; 17; 128]);;
42
17
128
- : unit = ()
```

List.rev [a1; ...; an] = [an; ... ;a1]

Reverse the order of the elements of a list.

Example: Enumerating List Elements

To transform a list and pass information between elements, use *List.fold_left* with a tuple:

```
# let (l, _) = List.fold_left
    (fun (l, n) e -> ((e, n)::l, n+1)) ([], 0) [42; 17; 128]
  in List.rev l;;
- : (int * int) list = [(42, 0); (17, 1); (128, 2)]
```

Can do the same with a recursive function.

```
# let rec enum n l =
   match l with
     | [] -> []
     | h :: t -> (h, n) :: enum (n + 1) t;;
val enum : int -> 'a list -> ('a * int) list = <fun>
# enum 0 [42; 17; 128];;
- : (int * int) list = [(42, 0); (17, 1); (128, 2)]
```

Example: Enumerating List Elements

Using tail recursion:

```
# let rec enum rl n l =
  match l with
    | [] -> List.rev rl
    | h :: t -> enum ((h, n) :: rl) (n + 1) t;;
val enum : ('a * int) list -> int -> 'a list -> ('a * int) list = <fun>
# enum [] 0 [42; 17; 128];;
- : (int * int) list = [(42, 0); (17, 1); (128, 2)]
```

Using a helper function:

```
# let enum l =
  let rec helper rl n l =
  match l with
    | [] -> List.rev rl
    | h :: t -> helper ((h, n) :: rl) (n + 1) t
  in helper [] 0 l;;
val enum : int -> 'a list -> ('a * int) list = <fun>
# enum [42; 17; 128];;
- : (int * int) list = [(42, 0); (17, 1); (128, 2)]
```

Pattern Matching

A powerful variety of multi-way branch that is adept at picking apart data structures. Unlike anything in C/C++/Java.

```
# let xor p =
match p  with
    | (false, false) -> false
    | (false,  true) -> true
    | (true, false) -> true
    | (true,  true) -> false;;
val xor : bool * bool -> bool = <fun>
# xor (true, true);;
- : bool = false
```

Pattern Matching

A name in a pattern matches anything and is bound when the pattern matches. Each may appear only once per pattern.

```
# let xor p =
match p with
    | (false, x) -> x
    | (true, x) -> not x;;
val xor : bool * bool -> bool = <fun>
# xor (true, true);;
- : bool = false
```

Case Coverage

The compiler warns you when you miss a case or when one is redundant (they are tested in order):

```
# let xor p = match p
  with (false, x) -> x
    | (x, true) -> not x;;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(true, false)
val xor : bool * bool -> bool = <fun>

# let xor p = match p
  with (false, x) -> x
    | (true,  x) -> not x
    | (false, false) -> false;;
Warning U: this match case is unused.
val xor : bool * bool -> bool = <fun>
```

Wildcards

Underscore (_) is a wildcard that will match anything, useful as a default or when you just don't care.

```
# let xor p = match p
  with (true, false) | (false, true) -> true
    | _ -> false;;
val xor : bool * bool -> bool = <fun>
# xor (true, true);;
- : bool = false
# xor (true, false);;
- : bool = true
# let logand p = match p
  with (false, _) -> false
    | (true, x) -> x;;
val logand : bool * bool -> bool = <fun>
# logand (true, false);;
- : bool = false
# logand (true, true);;
- : bool = true
```

Pattern Matching with Lists

```
# let length = function (* let length = fun p -> match p with *)
  | [] -> "empty"
  | [_] -> "singleton"
  | [_; _] -> "pair"
  | [_; _; _] -> "triplet"
  | hd :: tl -> "many";;
val length : 'a list -> string = <fun>

# length [];;
- : string = "empty"

# length [1; 2];;
- : string = "pair"

# length ["foo"; "bar"; "baz"];;
- : string = "triplet"

# length [1; 2; 3; 4];;
- : string = "many"
```

Pattern Matching with *when* and *as*

The *when* keyword lets you add a guard expression:

```
# let tall = function
  | (h, s) when h > 180 -> s ^ " is tall"
  | (_, s) -> s ^ " is short";;
val tall : int * string -> string = <fun>
# List.map  tall [(183, "Stephen"); (150, "Nina")];;
- : string list = ["Stephen is tall"; "Nina is short"]
```

The *as* keyword lets you name parts of a matched structure:

```
# match ([3;9], 4) with
  | (3::_ as xx, 4) -> xx
  | _ -> [];;
- : int list = [3; 9]
```

Application: Length of a list

```
let rec length l =
  if l = [] then 0 else 1 + length (List.tl l);;
```

Correct, but not very elegant. With pattern matching,

```
let rec length = function
 |   []     -> 0
 | _::tl -> 1 + length tl;;
```

Elegant, but inefficient because it is not tail-recursive (needs $O(n)$ stack space). Common trick: use an argument as an accumulator.

```
let length l =
  let rec helper len = function
    |   []     -> len
    | _::tl -> helper (len + 1) tl
  in helper 0 l
```

OCaml Can Compile This Efficiently

OCaml source code

```
let length list =
  let rec helper len = function
      []     -> len
    | _::tl -> helper (len + 1) tl
  in helper 0 list
```

- Arguments in registers
- Pattern matching reduced to a conditional branch
- Tail recursion implemented with jumps
- LSB of an integer always 1

ocamlopt generates this x86 assembly pseudocode

```
camlLength__helper:
.L101:
  cmpl  $1, %ebx       # empty?
  je    .L100
  movl  4(%ebx), %ebx  # get tail
  addl  $2, %eax       # len++
  jmp   .L101
.L100:
  ret

camlLength__length:
  movl  %eax, %ebx
  movl  $1, %eax       # len = 0
  jmp   camlLength__helper
```

User-Defined Types

Type Declarations

A new type name is defined globally. Unlike *let*, *type* is recursive by default, so the name being defined may appear in the *typedef*.

$$\text{type } \textit{name} = \textit{typedef}$$

Mutually-recursive types can be defined with *and*.

$$\begin{aligned} \text{type } \textit{name}_1 &= \textit{typedef}_1 \\ \text{and } \ \textit{name}_2 &= \textit{typedef}_2 \\ &\vdots \\ \text{and } \ \textit{name}_n &= \textit{typedef}_n \end{aligned}$$

Records

OCaml supports records much like C's *structs*.

```
# type base = { x : int; y : int; name : string };;
type base = { x : int; y : int; name : string; }
# let b0 = { x = 0; y = 0; name = "home" };;
val b0 : base = {x = 0; y = 0; name = "home"}
# let b1 = { b0 with x = 90; name = "first" };;
val b1 : base = {x = 90; y = 0; name = "first"}
# let b2 = { b1 with y = 90; name = "second" };;
val b2 : base = {x = 90; y = 90; name = "second"}
# b0.name;;
- : string = "home"
# let dist b1 b2 =
    let hyp x y = sqrt (float_of_int (x*x + y*y)) in
    hyp (b1.x - b2.x) (b1.y - b2.y);;
val dist : base -> base -> float = <fun>
# dist b0 b1;;
- : float = 90.
# dist b0 b2;;
- : float = 127.279220613578559
```

Algebraic Types/Tagged Unions/Sum-Product Types

Vaguely like C's *unions*, *enums*, or a class hierarchy: objects that can be one of a set of types. In compilers, great for trees and instructions.

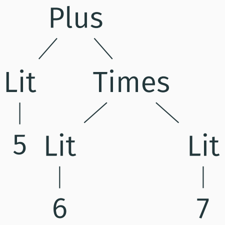
```
# type seasons = Winter | Spring | Summer | Fall;;
type seasons = Winter | Spring | Summer | Fall
# let weather = function
  | Winter -> "Too Cold"
  | Spring -> "Too Wet"
  | Summer -> "Too Hot"
  | Fall -> "Too Short";;
val weather : seasons -> string = <fun>

# weather Spring;;
- : string = "Too Wet"

# let year = [Winter; Spring; Summer; Fall] in
  List.map weather year;;
- : string list = ["Too Cold"; "Too Wet"; "Too Hot"; "Too Short"]
```

Simple Syntax Trees

```
# type expr =
    Lit of int
  | Plus of expr * expr
  | Minus of expr * expr
  | Times of expr * expr;;
type expr =
    Lit of int
  | Plus of expr * expr
  | Minus of expr * expr
  | Times of expr * expr
# Lit 42;;
- : expr = Lit 42
# Plus (Lit 5, Times (Lit 6, Lit 7));;
- : expr = Plus (Lit 5, Times (Lit 6, Lit 7))
```



Simple Syntax Trees and an Interpreter

```
# let rec eval = function
    Lit(x) -> x
  | Plus(e1, e2) -> (eval e1) + (eval e2)
  | Minus(e1, e2) -> (eval e1) - (eval e2)
  | Times(e1, e2) -> (eval e1) * (eval e2);;
val eval : expr -> int = <fun>
# eval (Lit(42));;
- : int = 42
# eval (Plus (Lit 5, Times (Lit 6, Lit 7)));;
- : int = 47
```

Algebraic Type Rules

Each tag name must begin with a capital letter

```
# let bad1 = left | right;;
Syntax error
```

Tag names must be globally unique (required for type inference)

```
# type weekend = Sat | Sun;;
type weekend = Sat | Sun
# type days = Sun | Mon | Tue;;
type days = Sun | Mon | Tue
# function Sat -> "sat" | Sun -> "sun";;
This pattern matches values of type days
but is here used to match values of type weekend
```

Algebraic Types and Pattern Matching

The compiler warns about missing cases:

```
# type expr =
    Lit of int
  | Plus of expr * expr
  | Minus of expr * expr
  | Times of expr * expr;;
type expr =
    Lit of int
  | Plus of expr * expr
  | Minus of expr * expr
  | Times of expr * expr
# let rec eval = function
    Lit(x) -> x
  | Plus(e1, e2) -> (eval e1) + (eval e2)
  | Minus(e1, e2) -> (eval e1) - (eval e2);;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Times (_, _)
val eval : expr -> int = <fun>
```

The *Option* Type: A Safe Null Pointer

Part of the always-loaded core library:

$$\text{type 'a option} = \text{None} \mid \text{Some of 'a}$$

This is a polymorphic algebraic type: 'a is any type. *None* is like a null pointer; *Some* is a non-null pointer. The compiler requires *None* to be handled explicitly.

```
# let rec sum = function
    []          -> 0                        (* base case *)
  | None::tl    -> sum tl  (* handle the "null pointer" case *)
  | Some(x)::tl -> x + sum tl;;               (* normal case *)
val sum : int option list -> int = <fun>

# sum [None; Some(5); None; Some(37)];;
- : int = 42
```

Algebraic Types vs. Classes and Enums

	Algebraic Types	Classes	Enums
Choice of Types	fixed	extensible	fixed
Operations	extensible	fixed	extensible
Fields	ordered	named	none
Hidden fields	none	supported	none
Recursive	yes	yes	no
Inheritance	none	supported	none
Case splitting	simple	costly	simple

An algebraic type is best when the set of types rarely change but you often want to add additional functions. Classes are good in exactly the opposite case.

Modules and Compilation

Modules

Each source file is a module and everything is public.

foo.ml

```
(* Module Foo *)

type t = { x : int ; y : int }
let sum c = c.x + c.y
```

bar.ml

```
(* The dot notation *)

let v = { Foo.x = 1 ;
          Foo.y = 2 };;
print_int (Foo.sum v)

(* Create a short name *)

module F = Foo;;
print_int (F.sum v)

(* Import every name from
   a module with "open" *)

open Foo;;
print_int (sum v)
```

To compile and run these,

```
$ ocamlc -c foo.ml
  (creates foo.cmi foo.cmo)
$ ocamlc -c bar.ml
  (creates bar.cmi bar.cmo)
$ ocamlc -o ex foo.cmo bar.cmo
$ ./ex
333
```

Separating Interface and Implementation

stack.mli

```
type 'a t

exception Empty

val create : unit -> 'a t
val push : 'a -> 'a t -> unit
val pop : 'a t -> 'a
val top : 'a t -> 'a
val clear : 'a t -> unit
val copy : 'a t -> 'a t
val is_empty : 'a t -> bool
val length : 'a t -> int
val iter : ('a -> unit) ->
                'a t -> unit
```

stack.ml

```
type 'a t =
   { mutable c : 'a list }
exception Empty
let create () = { c = [] }
let clear s = s.c <- []
let copy s = { c = s.c }
let push x s = s.c <- x :: s.c
let pop s =
  match s.c with
    hd::tl -> s.c <- tl; hd
  | []     -> raise Empty
let top s =
  match s.c with
    hd::_ -> hd
  | []    -> raise Empty
let is_empty s = (s.c = [])
let length s = List.length s.c
let iter f s = List.iter f s.c
```

Exceptions

Exceptions

```
# 5 / 0;;
Exception: Division_by_zero.

# try
    5 / 0
  with Division_by_zero -> 42;;
- : int = 42

# exception My_exception;;
exception My_exception
# try
    if true then
       raise My_exception
    else 0
  with My_exception -> 42;;
- : int = 42
```

Exceptions

```
# exception Foo of string;;
exception Foo of string
# exception Bar of int * string;;
exception Bar of int * string

# let ex b =
  try
    if b then
      raise (Foo("hello"))
    else
      raise (Bar(42, " answer"))
  with Foo(s) -> "Foo: " ^ s
  | Bar(n, s) -> "Bar: " ^ string_of_int n ^ s;;
val ex : bool -> unit = <fun>

# ex true;;
- : string = "Foo: hello"
# ex false;;
- : string = "Bar: 42 answer"
```

Standard Library Modules

Maps

Balanced trees for implementing dictionaries. Ask for a map with a specific kind of key; values are polymorphic.

```
# module StringMap = Map.Make(String);;
module StringMap :
  sig
    type key = String.t
    type 'a t = 'a Map.Make(String).t
    val empty : 'a t
    val is_empty : 'a t -> bool
    val add : key -> 'a -> 'a t -> 'a t
    val find : key -> 'a t -> 'a
    val remove : key -> 'a t -> 'a t
    val mem : key -> 'a t -> bool
    val iter : (key -> 'a -> unit) -> 'a t -> unit
    val map : ('a -> 'b) -> 'a t -> 'b t
    val mapi : (key -> 'a -> 'b) -> 'a t -> 'b t
    val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
    val compare : ('a -> 'a -> int) -> 'a t -> 'a t -> int
    val equal : ('a -> 'a -> bool) -> 'a t -> 'a t -> bool
  end
```

Maps

```
# let mymap = StringMap.empty;;          (* Create empty map *)
val mymap : 'a StringMap.t = <abstr>
# let mymap = StringMap.add "Douglas" 42 mymap;; (* Add pair *)
val mymap : int StringMap.t = <abstr>
# StringMap.mem "foo" mymap;;             (* Is "foo" there? *)
- : bool = false
# StringMap.mem "Douglas" mymap;;     (* Is "Douglas" there? *)
- : bool = true
# StringMap.find "Douglas" mymap;;               (* Get value *)
- : int = 42
# let mymap = StringMap.add "Adams" 17 mymap;;
val mymap : int StringMap.t = <abstr>
# StringMap.find "Adams" mymap;;
- : int = 17
# StringMap.find "Douglas" mymap;;
- : int = 42
# StringMap.find "Slarti" mymap;;
Exception: Not_found.
```

Maps

- Fully functional: *Map.add* takes a key, a value, and a map and returns a new map that also includes the given key/value pair.
- Needs a totally ordered key type. *Pervasives.compare* usually does the job (returns -1, 0, or 1); you may supply your own.

```
module StringMap = Map.Make(struct
  type t = string
  let compare x y = Pervasives.compare x y
end)
```

- Uses balanced trees, so searching and insertion is $O(\log n)$.

Imperative Features

```
# 0 ; 42;;                              (* ";" means sequencing *)
Warning S: this expression should have type unit.
- : int = 42
# ignore 0 ; 42;;          (* ignore is a function: 'a -> unit *)
- : int = 42
# () ; 42;;             (* () is the literal for the unit type *)
- : int = 42
# print_endline "Hello World!";;     (* Print; result is unit *)
Hello World!
- : unit = ()
# print_string "Hello " ; print_endline "World!";;
Hello World!
- : unit = ()
# print_int 42 ; print_newline ();;
42
- : unit = ()
# print_endline ("Hello " ^ string_of_int 42 ^ " world!");;
Hello 42 world!
- : unit = ()
```

Hash Tables

```
# module StringHash = Hashtbl.Make(struct
    type t = string                          (* type of keys *)
    let equal x y = x = y       (* use structural comparison *)
    let hash = Hashtbl.hash          (* generic hash function *)
  end);;
module StringHash :
  sig
    type key = string
    type 'a t
    val create : int -> 'a t
    val clear : 'a t -> unit
    val copy : 'a t -> 'a t
    val add : 'a t -> key -> 'a -> unit
    val remove : 'a t -> key -> unit
    val find : 'a t -> key -> 'a
    val find_all : 'a t -> key -> 'a list
    val replace : 'a t -> key -> 'a -> unit
    val mem : 'a t -> key -> bool
    val iter : (key -> 'a -> unit) -> 'a t -> unit
    val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
    val length : 'a t -> int
  end
```

Hash Tables

```
# let hash = StringHash.create 17;;  (* initial size estimate *)
val hash : '_a StringHash.t = <abstr>
# StringHash.add hash "Douglas" 42;; (* modify the hash table *)
- : unit = ()
# StringHash.mem hash "foo";;           (* is "foo" there? *)
- : bool = false
# StringHash.mem hash "Douglas";;     (* is "Douglas" there? *)
- : bool = true
# StringHash.find hash "Douglas";;              (* Get value *)
- : int = 42
# StringHash.add hash "Adams" 17;;   (* Add another key/value *)
- : unit = ()
# StringHash.find hash "Adams";;
- : int = 17
# StringHash.find hash "Douglas";;
- : int = 42
# StringHash.find hash "Slarti";;
Exception: Not_found.
```

Arrays

```
# let a = [| 42; 17; 19 |];;              (* Array literal *)
val a : int array = [|42; 17; 19|]
# let aa = Array.make 5 0;;               (* Fill a new array *)
val aa : int array = [|0; 0; 0; 0; 0|]
# a.(0);;                                 (* Random access *)
- : int = 42
# a.(2);;
- : int = 19
# a.(3);;
Exception: Invalid_argument "index out of bounds".
# a.(2) <- 20;;                           (* Arrays are mutable! *)
- : unit = ()
# a;;
- : int array = [|42; 17; 20|]
# let l = [24; 32; 17];;
val l : int list = [24; 32; 17]
# let b = Array.of_list l;;               (* Array from a list *)
val b : int array = [|24; 32; 17|]
# let c = Array.append a b;;              (* Concatenation *)
val c : int array = [|42; 17; 20; 24; 32; 17|]
```

Arrays vs. Lists

	Arrays	**Lists**
Random access	$O(1)$	$O(n)$
Appending	$O(n)$	$O(1)$
Mutable	Yes	No

Useful pattern: first collect data of unknown length in a list then convert it to an array with *Array.of_list* for random queries.

A Complete Interpreter in Three Slides

The Scanner and AST

scanner.mll

```
{ open Parser }

rule token =
  parse [' ' '\t' '\r' '\n'] { token lexbuf }
      | '+'                  { PLUS }
      | '-'                  { MINUS }
      | '*'                  { TIMES }
      | '/'                  { DIVIDE }
      | ['0'-'9']+ as lit    { LITERAL(int_of_string lit) }
      | eof                  { EOF }
```

ast.mli

```
type operator = Add | Sub | Mul | Div

type expr =
    Binop of expr * operator * expr
  | Lit of int
```

The Parser

parser.mly

```
%{ open Ast %}

%token PLUS MINUS TIMES DIVIDE EOF
%token <int> LITERAL

%left PLUS MINUS
%left TIMES DIVIDE

%start expr
%type <Ast.expr> expr

expr:
  expr PLUS   expr   { Binop($1, Add, $3) }
| expr MINUS  expr   { Binop($1, Sub, $3) }
| expr TIMES  expr   { Binop($1, Mul, $3) }
| expr DIVIDE expr   { Binop($1, Div, $3) }
| LITERAL            { Lit($1) }
```

The Interpeter

calc.ml

```
open Ast

let rec eval = function
    Lit(x) -> x
  | Binop(e1, op, e2) ->
      let v1 = eval e1 and v2 = eval e2 in
      match op with
        Add -> v1 + v2
      | Sub -> v1 - v2
      | Mul -> v1 * v2
      | Div -> v1 / v2

let _ =
  let lexbuf = Lexing.from_channel stdin in
  let expr = Parser.expr Scanner.token lexbuf in
  let result = eval expr in
  print_endline (string_of_int result)
```

Compiling the Interpreter

```
$ ocamllex scanner.mll # create scanner.ml
8 states, 267 transitions, table size 1116 bytes
$ ocamlyacc parser.mly # create parser.ml and parser.mli
$ ocamlc -c ast.mli    # compile AST types
$ ocamlc -c parser.mli # compile parser types
$ ocamlc -c scanner.ml # compile the scanner
$ ocamlc -c parser.ml  # compile the parser
$ ocamlc -c calc.ml    # compile the interpreter
$ ocamlc -o calc parser.cmo scanner.cmo calc.cmo
$ ./calc
2 * 3 + 4 * 5
26
$
```

Compiling with *ocamlbuild*

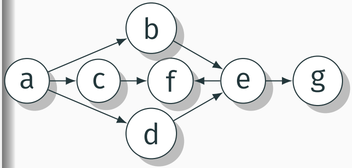
```
$ ls
ast.mli  calc.ml  parser.mly  scanner.mll
$ ocamlbuild calc.native   # Build everything
Finished, 15 targets (0 cached) in 00:00:00.
$ ls
ast.mli _build calc.ml calc.native parser.mly scanner.mll
$ ./calc.native
2 * 3 + 4 * 5
Ctrl-D
26
 ocamlbuild -clean # Remove _build and all .native
```

Directed Graphs

Application: Directed Graphs

```
let edges = [
  ("a", "b"); ("a", "c");
  ("a", "d"); ("b", "e");
  ("c", "f"); ("d", "e");
  ("e", "f"); ("e", "g") ]

let rec successors n = function
    []                -> []
  | (s, t) :: edges ->
      if s = n then
        t :: successors n edges
      else
        successors n edges
```




```
# successors "a" edges;;
- : string list = ["b"; "c"; "d"]

# successors "b" edges;;
- : string list = ["e"]
```

More Functional Successors

```
let rec successors n = function
    []                -> []
  | (s, t) :: edges ->
      if s = n then
        t :: successors n edges
      else
        successors n edges
```

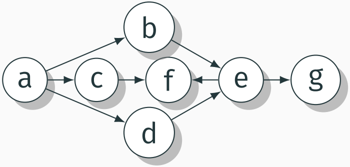
Our first example is imperative: performs “search a list,” which is more precisely expressed using the library function List.filter:

```
let successors n edges =
    let matching (s,_) = s = n in
    List.map snd (List.filter matching edges)
```

This uses the built-in snd function, which is defined as

```
let snd (_,x) = x
```

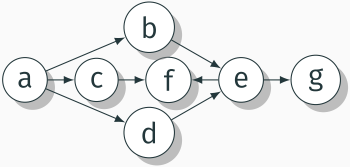
Depth-First Search




```
let rec dfs edges visited = function
  []        -> List.rev visited
| n::nodes ->
  if List.mem n visited then
    dfs edges visited nodes
  else
    dfs edges (n::visited) ((successors n edges) @ nodes)
```

```
# dfs edges [] ["a"];;
- : string list = ["a"; "b"; "e"; "f"; "g"; "c"; "d"]
# dfs edges [] ["e"];;
- : string list = ["e"; "f"; "g"]
# dfs edges [] ["d"];;
- : string list = ["d"; "e"; "f"; "g"]
```

Topological Sort



Remember the visitor at the end.

```
let rec tsort edges visited = function
  []          -> visited
| n::nodes ->
  let visited' = if List.mem n visited then visited
                 else n :: tsort edges visited (successors n
  in tsort edges visited' nodes;;
```

```
# tsort edges [] ["a"];;
- : string list = ["a"; "d"; "c"; "b"; "e"; "g"; "f"]
# let cycle = [ ("a", "b"); ("b", "c"); ("c", "a") ];;
val cycle : (string * string) list = [("a", "b"); ...]
# tsort cycle [] ["a"];;
Stack overflow during evaluation (looping recursion?).
```

Better Topological Sort

```
exception Cyclic of string
let tsort edges seed =
  let rec sort path visited = function
      []         -> visited
    | n::nodes ->
      if List.mem n path then raise (Cyclic n) else
      let v' = if List.mem n visited then visited else
                 n :: sort (n::path) visited (successors n edge
      in sort path v' nodes
  in
  sort [] [] [seed]
```

```
# tsort edges "a";;
- : string list = ["a"; "d"; "c"; "b"; "e"; "g"; "f"]
# tsort edges "d";;
- : string list = ["d"; "e"; "g"; "f"]
# tsort cycle "a";;
Exception: Cyclic "a".
```

Depth-First Search Revisited

Previous version

```
let rec dfs edges visited = function
  []          -> List.rev visited
| n::nodes ->
  if List.mem n visited then
    dfs edges visited nodes
  else
    dfs edges (n::visited) ((successors n edges) @ nodes)
```

was not very efficient, but good enough for small graphs.

Would like faster *visited* test and *successors* query.

Depth-First Search Revisited

Second version:

- use a Map to hold a list of successors for each node
- use a Set (valueless Map) to remember of visited nodes

```
module StringMap = Map.Make(String)
module StringSet = Set.Make(String)
```

Depth-First Search Revisited

```
let top_sort_map edges =
  (* Create an empty successor list for each node *)
  let succs = List.fold_left
      (fun map (s,d) ->
        StringMap.add d [] (StringMap.add s [] map)
      ) StringMap.empty edges
  in (* Build the successor list for each source node *)
  let succs = List.fold_left
      (fun succs (s, d) ->
        let ss = StringMap.find s succs
        in StringMap.add s (d::ss) succs) succs edges
  in
  (* Visit recursively, storing each node after visiting successors*)
  let rec visit (order, visited) n =
    if StringSet.mem n visited then
      (order, visited)
    else  let (order, visited) = List.fold_left
                visit (order, StringSet.add n visited)
                (StringMap.find n succs)
          in (n::order, visited)
  in (* Visit the source of each edge *)
  fst (List.fold_left visit ([], StringSet.empty)
                            (List.map fst edges))
```

DFS with Arrays

Second version used a lot of *mem*, *find*, and *add* calls on the string map, each $O(\log n)$. Can we do better?

Solution: use arrays to hold adjacency lists and track visiting information.

Basic idea: number the nodes, build adjacency lists with numbers, use an array for tracking visits, then transform back to list of node names.

DFS with Arrays 1/2

```
let top_sort_array edges =
  (* Assign a number to each node *)
  let map, nodecount =
    List.fold_left
      (fun nodemap (s, d) ->
        let addnode node (map, n) =
          if StringMap.mem node map then (map, n)
          else (StringMap.add node n map, n+1)
        in
        addnode d (addnode s nodemap)
      ) (StringMap.empty, 0) edges
  in

  let successors = Array.make nodecount [] in
  let name = Array.make nodecount "" in

  (* Build adjacency lists and remember the name of each node *)
  List.iter
    (fun (s, d) ->
      let ss = StringMap.find s map in
      let dd = StringMap.find d map in
      successors.(ss) <- dd :: successors.(ss);
      name.(ss) <- s;
      name.(dd) <- d;
    ) edges;
```

DFS with Arrays 2/2

```
(* Visited flags for each node *)
let visited = Array.make nodecount false in

(* Visit each of our successors if we haven't done so yet *)
(* then record the node *)
let rec visit order n =
  if visited.(n) then order
  else (
    visited.(n) <- true;
    n :: (List.fold_left visit order successors.(n))
  )
in

(* Compute the topological order *)
let order = visit [] 0 in

(* Map node numbers back to node names *)
List.map (fun n -> name.(n)) order
```